Introducción a la Programación
IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Programación Orientada a Objetos
Parte 3

Colaboración de Valeria Herskovic

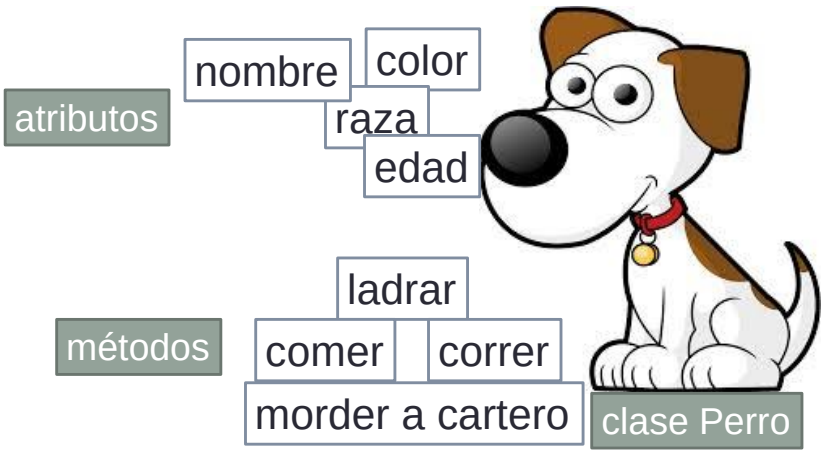
Agenda

1. Objetos y Clases
2. Métodos especiales:
   a. __init__
   b. __str__
   c. Overloading (“Sobre-carga de operadores”)
3. Comunicación entre objetos
4. Ejemplos

Recordemos

- Clases: atributos + métodos

Los objetos Bobby, Arya y Pepita son instancias de la clase Perro.



Pero, ¿porqué usar POO?

Todos los problemas, ejercicios y tareas que hemos visto hasta ahora en el curso, los hemos resuelto correctamente con programación "**funcional**" (variables y funciones con parámetros), sin objetos.

La **POO** nos servirá para resolver las mismas situaciones pero desde la perspectiva de los "**objetos**". Esto nos da una mayor cercanía al "mundo real". Y para problemas de mayor tamaño y complejidad, la **POO** será más eficiente, compacta, integrada, entendible, corregible y facilitará la construcción "evolutiva" de programas (más re-usabilidad).

Básicamente, un **POO** se construye definiendo (**clases** de) **objetos** y funciones que operan sobre ellos, las cuales "encapsulamos" como **métodos** "dentro" de la clase. La **interacción** entre los objetos se realiza a través de sus **métodos**.
Igual podemos seguir usando instrucciones de la programación funcional, tales como variables, condicionales y ciclos.

Métodos Especiales para Clases

Ya vimos el método "constructor" **__init__()** que se ejecuta cuando un objeto es creado ("instanciado"), dando valores iniciales a los atributos del nuevo objeto.

Otro método especial es **__str__()** el cual retorna la representación de un objeto en forma de string. Lo podemos usar para imprimir con **print()**:

print (nombre_objeto**)**

Si no ocupamos este método, tenemos que hacer:

print (objeto.atrib1, objeto.atrib2, objeto.atrib3, …**)**

Nota: Si utilizamos **str (nombre_objeto)** se invoca al método **__str__()** y retorna un string.

Métodos Especiales para Clases

Veamos un ejemplo:

```
class Time:
    """ Clase para representar objetos
        con atributos Hora, Minuto, Segundo
    """
    def __init__(self, hora=0, mins=0, segs=0):
        self.hora = hora
        self.mins = mins
        self.segs = segs
    def __str__(self):
        return(str(self.hora)+":"+str(self.mins)
               +":"+str(self.segs))
# programa ppal
Tinicial = Time()
print("Inicial: ", Tinicial)
Tahora = Time(8,55,15)
print("Ahora: ", Tahora)
```

Nota:
print() invoca al método **__str__**

El objeto **Tahora** sigue siendo de clase **Time**.

```
>>>
Inicial:  0:0:0
Ahora:  8:55:15
>>>
```

Otros Métodos Especiales: Overloading "Sobrecarga de Operadores"

Podemos utilizar métodos especiales de "**sobrecarga**" para "extender" los operadores pre-definidos en Python a operaciones con objetos.

Por ejemplo, si queremos sumar objetos de clase **Time** (ejemplo anterior), utilizamos el método especial:

__add__()

Dentro de la definición de la clase **Time**, definimos la operación como nosotros queremos:

```
def __add__(self, otro):
    n = Time()
    n.hora = self.hora + otro.hora
    n.mins = self.mins + otro.mins
    n.segs = self.segs + otro.segs
    return (n)
```

Métodos de Overloading

Veamos el ejemplo con la clase Time:

```
class Time:
    """ Atributos Hora, Minuto, Segundo """
    def __init__(self, hora=0, mins=0, segs=0):
        self.hora = hora
        self.mins = mins
        self.segs = segs
    def __str__(self):
        return(str(self.hora)+":"+str(self.mins)
               +":"+str(self.segs))
    def __add__(self, otro):
        n = Time()
        n.hora = self.hora + otro.hora
        n.mins = self.mins + otro.mins
        n.segs = self.segs + otro.segs
        return (n)
# programa ppal
Inicio = Time(8,30)
print("Inicio: ", Inicio)
Durac = Time(1,20,30)
print("Durac: ", Durac)
print("Término: ", Inicio + Durac)
```

```
>>>
Inicio:  8:30:0
Durac:  1:20:30
Término:  9:50:30
```

Nota 1:
El método **__add__** no modifica a los operandos **Inicio** y **Durac**.

Nota 2:
Para cada operación pre-definida en Python, existe un **método de overloading** de objetos.

Problema con el ejemplo anterior:

¿Qué pasa si los mins y/o segs suman más de 60? Una solución:

```
class Time:
    """ Atributos Hora, Minuto, Segundo """
    def __init__(self, hora=0, mins=0, segs=0):
        self.hora = hora
        self.mins = mins
        self.segs = segs
    def __str__(self):
        return(str(self.hora)+":"+str(self.mins)
               +":"+str(self.segs))
    def time_a_int(self):
        minutos = self.hora * 60 + self.mins
        segundos = minutos * 60 + self.segs
        return (segundos)
    def __add__(self, otro):
        segundos = self.time_a_int() + otro.time_a_int()
        return (int_a_time(segundos))
def int_a_time (num):
        t = Time()
        minutos, t.segs = divmod (num, 60)
        t.hora, t.mins = divmod (minutos, 60)
        return (t)
# programa ppal
Inicio = Time(8,30,30)
print("Inicio: ", Inicio, " en segundos: ", Inicio.time_a_int())
Durac = Time(1,50,50)
print("Durac: ", Durac)
print("Término: ", Inicio + Durac)
```

Nota: Esta función está fuera de la clase Time. Por eso no se invoca como método. (La indentación está OK.)

```
>>>
Inicio:  8:30:30  en segundos:  30630
Durac:  1:50:50
Término:  10:21:20
>>>
```

Otros Métodos Especiales: Overloading "Sobrecarga de Operadores"

Para resumir:
podemos utilizar métodos especiales de "**sobrecarga**" para "extender" los operadores pre-definidos en Python a operaciones con objetos.

Para sumar objetos: **__add__()**
Para restar objetos: **__sub__()**
Para multiplicar objetos: **__mul__()**
Para dividir objetos: **__truediv__()** **__floordiv()__** **__divmod__()**
Para potencias: **__pow__()**

Para operaciones lógicas (booleans):
__lt__() **__le__()** **__eq__()** **__ne__()** **__gt__()** **__ge__()**

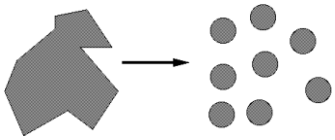
__and__() **__or__()** **__xor__()**

¿Cómo interactúan los Objetos entre sí?

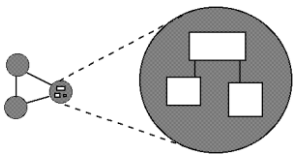
A través de sus **métodos**

- `bobby = Perro("Bobby", "quiltro",3, "blanco")`
- `juan = Cartero("Juan Perez")`
- `bobby.ladrar()`
- `bobby.ladrar()`
- `bobby.morder(juan)`

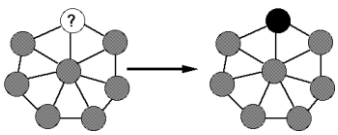
Programación Orientada a Objetos- por qué? (algunas razones)



Modularización: descomponer problemas grandes en problemas más pequeños.



Abstracción/Entendimiento: terminología del dominio del problema está en la solución. Módulos individuales son entendibles.

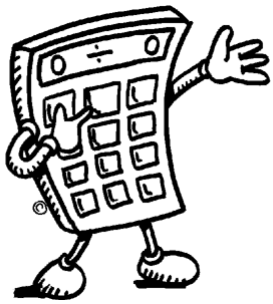


Encapsulamiento – escondemos detalles de posibles malos usos por los usuarios .

ref: http://www.felixgers.de/teaching/oop/oop_intro.html

Ejemplo #1: Calculadora de Fracciones

- fraccion 1? **"3/4"**
- fraccion 2? **"6/8"**
- suma = 48/32 = 3/2



Solución usando POO ☺

Nos gustaría poder hacer lo siguente:

```
f1 = Fraccion(input("fraccion 1? "))

f2 = Fraccion(input("fraccion 2? "))

f = f1.suma(f2)  # idealmente: f = f1 + f2

print( "suma =" + f + " =" + f.simple())
```

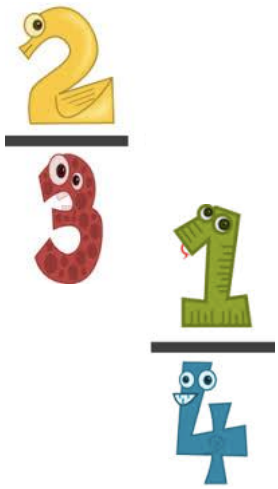
```
>>> suma = 48/32 = 3/2
```

clase Fraccion: atributos + metodos

atributos

a

b



métodos

suma

simple

str

Definiendo nuestra clase Fraccion

- f1 = Fraccion("3/4")

f1 →

a	3
b	4

Cada fracción tiene dos variables internas o atributos, llamados a y b

- f2 = Fraccion(6,3)

f2 →

a	6
b	3

clase Fraccion

(object) es opcional

```
class Fraccion(object): # Nueva clase Fraccion
    """Representa una fraccion
       atributos: a, b"""
    def __init__(Fraccion ,….):
        …
    def simple(Fraccion):
        …
    def suma(Fraccion, Fraccion):
        …

f1 = Fraccion()
```

indentación aclara que método pertenece a objeto

clase Fraccion

(object) es opcional

```
class Fraccion(object): #Nueva clase Fraccion
    """Representa una fraccion"""
    """atributos: a, b"""

    def __init__(self,…):    # Constructor
    …
    def simple(self):
    …
    def suma(self, other):
  …
# Programa ppal
f1 = Fraccion(input("fraccion 1? "))
```

self: objeto sobre el cual se aplica la operación

Creando fracciones…

```
class Fraccion(object):
    """Representa una fraccion
       atributos: a (numerador), b (denominador)"""
    def __init__(self, a=0, b=1):
        if isinstance(a,str):
            self.a = int(a[:a.index("/")])
            self.b = int(a[a.index("/")+1:])
        else:
            self.a = a
            self.b = b

frac1 = Fraccion(3,4)
frac2 = Fraccion(5)
frac3 = Fraccion()
frac= Fraccion("3/4")
```

Simplificando fracciones

Función que retorna el máximo común divisor. Se asume que la tenemos definida fuera de la clase.

```
class Fraccion(object):
    ……
    def simple(self):
        nueva=Fraccion()
        nueva.a = self.a//mcd(self.a,self.b)
        nueva.b = self.b//mcd(self.a,self.b)
        return nueva

frac = Fraccion(10,2)
# notación funcional:
simplificada = Fraccion.simple(frac)
# notación POO:
simplificada = frac.simple()
```

son equivalentes

Sumando fracciones

```
def suma(self, other):
        nueva=Fraccion()
        nueva.a = self.a*other.b+self.b*other.a
        nueva.b = self.b*other.b
        return nueva
```

```
f1 = Fraccion(3,4)
f2 = Fraccion(6,8)
f3 = f1.suma(f2)
```

Este método recibe dos objetos:
- self: sobre el cual se aplica la operación
- other: segundo parámetro

Recordemos:

Nos gustaría hacer lo siguiente:

```
f1 = Fraccion(input("fraccion 1? "))
f2 = Fraccion(input("fraccion 2? "))
f = f1+f2  #en vez de f1.suma(f2)
print("suma="+ f + "=" + f.simple())
```

Solución #1: suma

```
def __add__ (self,other): #operator overloading
        nueva=Fraccion()
        nueva.a = self.a*other.b+self.b*other.a
        nueva.b = self.b*other.b
        return nueva
```

__add__ será llamada con el operador +:
ej: f = f1+f2
es lo mismo que
f = f1.__add__(f2)

Pero: ¿Qué pasa si digo:
```
f1 = Fraccion(3,4)
f = f1+4
```
?

Solución #2: suma

```
def __add__(self,other): #operator overloading
        nueva=Fraccion()
        if isinstance(other,Fraccion):
            nueva.a = self.a*other.b+self.b*other.a
            nueva.b = self.b*other.b
        else: #es entero
            nueva.a = self.a + other*self.b
            nueva.b = self.b
        return nueva
```

Pero: ¿Qué pasa si digo:
```
f1 = Fraccion(3,4)
f = 4+f1
```
?

Solución #2b: suma

```
def __add__ (self,other): #operator overloading
      nueva=Fraccion()
      if isinstance(other,Fraccion):
          nueva.a = self.a*other.b+self.b*other.a
          nueva.b = self.b*other.b
      else: #es entero
          nueva.a = self.a + other*self.b
          nueva.b = self.b
      return nueva
def __radd__(self,other): #right-side add: x + Fraccion
      return self.__add__(other)
```

f = 4+f1
f1 es self
4 es other

¿Qué más hay?

```
def __str__(self):
       if self.a % self.b ==0:
           return str(self.a//self.b)
       else:
           return str(self.a)+"/"+str(self.b)
```

__str__ es la función que será llamada cuando pidamos str(x) para una Fraccion x

¿Más Overloading?

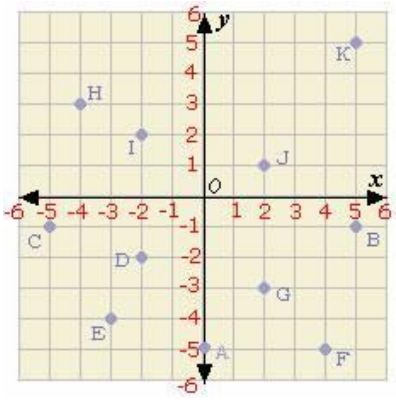
http://docs.python.org/3.1/reference/datamodel.html

- object.__lt__(self, other) # <
- object.__le__(self, other) # <=
- object.__eq__(self, other) # ==
- object.__ne__(self, other) # !=
- object.__gt__(self, other) # >
- object.__ge__(self, other) # >=
- object.__len__(self)
- object.__add__(…)
- object.__radd__(…)
- object.__iadd__(…)
- object.__mul__(…)
- object.__rmul__(…)
- object.__imul__(…)

Problema Propuesto.

Crear una clase que represente puntos de un plano cartesiano

```
a = Punto2d(0,-5)
k = Punto2d(5,5)
dist = a.distancia(k)
print("La distancia es"
      +str(dist))
c = a + k
```



Completar – propuesto para la casa

```
import math
class Punto2d(object):
    """ Representa un punto 2D en un plano
        cartesiano
        attributes: …."""
    def __init__(…) :
        …
    def distancia(…):
        …
```

Introducción a la Programación
IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Programación Orientada a Objetos
Parte 2

Colaboración de Valeria Herskovic